UMass Boston Computer Science

CS450 High Level Languages (section 2)

Intertwined Data

Wednesday, November 1, 2023

THE GOVERNMENT FINALLY DECIDES TO PUT AN END TO ALL THE ARGUMENTS.

Logistics

- HW 5 out
  - **UPDATE**: split into two parts
  - ~~Part 1 due: **Sun 10/29 11:59 pm EST**~~
  - Part 2 due: **Sun 11/5 11:59 pm EST**

- (Daylight Saving ends 11/5)

THE GOVERNMENT FINALLY DECIDES TO PUT AN END TO ALL THE ARGUMENTS.

Finding a Value in a Tree?

- Do we have to search the entire tree?

Data Definitions With Invariants

```
;; A Tree<X> is one of:
;; - empty
;; - (node Tree<X> X Tree<X>)
(struct node [left data right])
;; a binary tree data structure
```

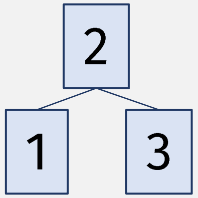
Predicate?

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀x ∈left tree, x < node-data
;; Invariant 2: ∀y ∈right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

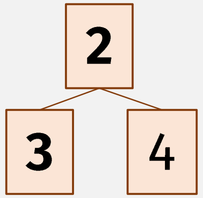
Valid BSTs

```
;; valid-bst? : Tree<X> -> Bool
;; Returns true if the given tree is a BST
```

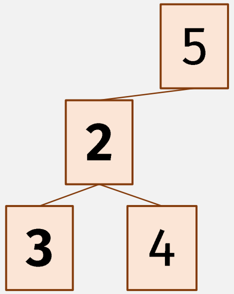
Valid



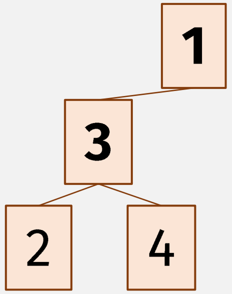
Not Valid



left value > root ☒



left values less than root ☑,
but left subtree not BST ☒



Left subtree is valid BST ☑,
but left values not less than root ☒

Valid BSTs

```
;; valid-bst? : Tree<X> -> Bool
;; Returns true if the tree is a BST
(define (valid-bst? t)
  (cond
    [(empty? t) true]
    [(node? t)
     (and (tree-all? (curry > (node-data t)) (node-left t))
          (tree-all? (curry <= (node-data t)) (node-right t))
          (valid-bst? (node-left t))
          (valid-bst? (node-right t)))]))
```

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀ x ∈ left tree, x < node-data
;; Invariant 2: ∀ y ∈ right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

cond that evaluates to a boolean is just boolean arithmetic!

```
(define (valid-bst? t)
  (or (empty? t)
      (and (tree-all? (curry > (node-data t)) (node-left t))
           (tree-all? (curry <= (node-data t)) (node-right t))
           (valid-bst? (node-left t))
           (valid-bst? (node-right t)))))
```

One-pass `valid-bst?`

```
;; valid-bst/one-pass? : Tree<X> -> Bool
;; Returns true if the tree is a BST
```

```
(define (valid-bst/one-pass? t)
  (or (empty? t)
      (and (valid-bst/one-pass? (node-left t))
           (valid-bst/one-pass? (node-right t)))))
```

One-pass `valid-bst?`

```
;; valid-bst/one-pass? : ??? Tree<X> -> Bool
;; Returns true if the tree is a BST
```

```
(define (valid-bst/one-pass? ??? t)
  (or (empty? t)
      (and (valid-bst/one-pass? ??? ??? (node-left t))
           (valid-bst/one-pass? ??? ??? (node-right t)))))
```

- Need extra argument(s) …
- … to keep track of valid interval for node-data value

One-pass **valid-bst?**, Functional Style!

```
;; valid-bst/p? : Tree<X> (X -> Bool) -> Bool
;; Returns true if (p? (node-data t)) = true, and t is a BST
```

```
(define (valid-bst/p? p? t)
  (or (empty? t)
      (and (p? (node-data t))
           (valid-bst/p? ???
                                 (node-left t))
           (valid-bst/p? ???
                                 (node-right
```

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀x ∈ left tree, x < node-data
;; Invariant 2: ∀y ∈ right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

One-pass **`valid-bst?`**, Functional Style!

```
;; valid-bst/p? : Tree<X> (X -> Bool) -> Bool
;; Returns true if (p? (node-data t)) = true, and t is a BST
```

```
(define (valid-bst/p? p? t)
  (or (empty? t)
      (and (p? (node-data t))
           (valid-bst/p?

                                 (curry > (node-data t)))
                        (node-left t))
           (valid-bst/p? ???



                        (node-right
```

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀x ∈left tree, x < node-data
;; Invariant 2: ∀y ∈right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

One-pass **valid-bst?**, Functional Style!

```
;; valid-bst/p? : Tree<X> (X -> Bool) -> Bool
;; Returns true if (p? (node-data t)) = true, and t is a BST
```

```
(define (valid-bst/p? p? t)
  (or (empty? t)
      (and (p? (node-data t))
           (valid-bst/p? (lambda (x)
                           (and (p? x)
                                ((curry > (node-data t)) x)))
                         (node-left t))
           (valid-bst/p? ???



                         (node-right
```

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀ x ∈ left tree, x < node-data
;; Invariant 2: ∀ y ∈ right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

One-pass **valid-bst?**, Functional Style!

```
;; valid-bst/p? : Tree<X> (X -> Bool) -> Bool
;; Returns true if (p? (node-data t)) = true, and t is a BST
```

```
(define (valid-bst/p? p? t)
  (or (empty? t)
      (and (p? (node-data t))
           (valid-bst/p? (lambda (x)
                           (and (p? x)
                                ((curry > (node-data t)) x)))
                         (node-left t))
           (valid-bst/p? (lambda (x)
                           (and (p? x)
                                ((curry <= (node-data t)) x)))
                         (node-right t)))))
```

```
(conjoin p1? p2?)
==
(λ (x) (and (p1? x) (p2? x)))
```

"conjoin" combines predicates

One-pass **valid-bst?**, Functional Style!

```
;; valid-bst/p? : Tree<X> (X -> Bool) -> Bool
;; Returns true if (p? (node-data t)) = true, and t is a BST
```

```
(define (valid-bst/p? p? t)
  (or (empty? t)
      (and (p? (node-data t))
           (valid-bst/p? (conjoin
                            p?
                            (curry > (node-data t))    )
                         (node-left t))
           (valid-bst/p? (conjoin
                            p?
                            (curry <= (node-data t))    )
                         (node-right t)))))
```

```
(conjoin p1? p2?)
==
(λ (x) (and (p1? x) (p2? x)))
```

```
(define (valid-bst? t)
  (valid-bst/p? (lambda (x) true) t))
```

Data Definitions With Invariants

Predicate?

```
;; A Tree<X> is one of:
;; - empty
;; - (node Tree<X> X Tree<X>)
(struct node [left data right])
;; a binary tree data structure
```

BST contracts should use "shallow" tree? predicate, not "deep" valid-bst?

```
(define (tree? x)
  (or (empty? x) (node? x)))
```

"Deep" Invariants are enforced by each BST function

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀x ∈left tree, x < node-data
;; Invariant 2: ∀y ∈right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

BST Insert

Must preserve BST invariants

Hint: use `valid-bst?` For tests

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define TREE2 (node empty 2 empty))
(define TREE123 (node TREE1 2 TREE3))
```

```
(check-equal? (bst-insert (bst-insert TREE2 1) 3)
              TREE123))
```

```
(check-true (valid-bst? (bst-insert TREE123 4)))
```

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< (node-data bst))
         (node (bst-insert (node-left t) x)
               (node-data t)
               (node-right t))
         (node (node-left t)
               (node-data t)
               (bst-insert (node-right t) x)))]))
```

Template:
cond clause for each itemization item

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< (node-data bst))
         (node (bst-insert (node-left t) x)
               (node-data t)
               (node-right t))
         (node (node-left t)
               (node-data t)
               (bst-insert (node-right t) x)))]))
```

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< (node-data bst))
         (node (bst-insert (node-left t) x)
               (node-data t)
               (node-right t))
         (node (node-left t)
               (node-data t)
               (bst-insert (node-right t) x)))]))
```

Template:
Recursive call matches recursion in data definition

Template:
Extract pieces of compound data

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< x (node-data bst))
         (node (bst-insert (node-left t) x)
               (node-data t)
               (node-right t))
         (node (node-left t)
               (node-data t)
               (bst-insert (node-right t) x)))]))
```

Result must maintain **BST invariant!**

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< x (node-data bst))
         (node (bst-insert (node-left t) x)
               (node-data t)
               (node-right t))
         (node (node-left t)
               (node-data t)
               (bst-insert (node-right t) x)))]))
```

Result must maintain **BST invariant!**

Smaller values on left

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< (node-data bst))
         (node (bst-insert (node-left t) x)
               (node-data t)
               (node-right t))
         (node (node-left t)
               (node-data t)
               (bst-insert (node-right t) x)))]))
```

Result must maintain **BST invariant!**

Larger values on right

Last Time

Finding a Value in a Tree?

- Do we have to search the entire tree?

Finding a Value in a BST?

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

```
(define TREE1 (node empty 1 empty))
(define TREE3 (node empty 3 empty))
(define TREE123 (node TREE1 2 TREE3))
```

```
(check-true (valid-bst? TREE123))
```

```
(check-true (bst-has? TREE123 1))
(check-false (bst-has? TREE123 4))
```

```
(check-true (bst-has? (bst-insert TREE123 4) 4))
```

Finding a Value in a BST?

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

```
(define (bst-has? bst x)
       ???  (empty? bst)
       ???           (node-data bst)
       ??? (bst-has? (node-left t) x)
       ??? (bst-has? (node-right t) x) )
```

BST (bool result) **Template**

Finding a Value in a BST?

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       ???            (node-data bst)
       ??? (bst-has? (node-left t) x)
       ??? (bst-has? (node-right t) x) )
```

BST cannot be empty

Finding a Value in a BST?

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       (or (equal? x (node-data bst))
       ??? (bst-has? (node-left t) x)
       ??? (bst-has? (node-right t) x) )
```

Either:

- (node-data bst) is x

Finding a Value in a BST?

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       (or (equal? x (node-data bst))
           (bst-has? (node-left t) x)
       ??? (bst-has? (node-right t) x) )
```

Either:
- (node-data bst) is x
- left subtree has x

Finding a Value in a BST?

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       (or (equal? x (node-data bst))
           (bst-has? (node-left t) x)
           (bst-has? (node-right t) x))))
```

Either:
- (node-data bst) is x
- left subtree has x
- right subtree has x

Finding a Value in a BST?

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       (or (equal? x (node-data bst))
           (bst-has? (node-left t) x)
           (bst-has? (node-right t) x))))
```

and and **or** are "short circuiting"
(stop search as soon as **x** is found)

Intertwined Data Definitions

- Come up with a Data Definition for …
- … valid Racket Programs

Valid Racket Programs

- 1
- “one”
- (+ 1 2)

```
;; A RacketProg is a:
;; - Number
;; - String
;; - ???
```

Valid Racket Programs

- 1
- “one”
- (+ 1 2)

```
;; A RacketProg is a:
;; - Atom
```

```
;; - ???
```

```
;; An Atom is a:
;; - Number
;; - String
```

Valid Racket Programs

- (+ 1 2)

List of … atoms?

"symbol"

```
;; A RacketProg is a:
;; - Atom
;; - List<Atom> ???
```

```
;; An Atom is a:
;; - Number
;; - String
;; - Symbol
```

Valid Racket Programs

- `(* (+ 1 2)`
  `   (- 4 3))` ← Tree?
- `(* (+ 1 2)`
  `   (- 4 3)`
  `   (/ 10 5))`

Each tree "node" is a list, of … RacketProgs ??

But: how many values does each node have??

```
;; A RacketProg is a:
;; - Atom
;; - Tree<???>
```

```
;; An Atom is a:
;; - Number
;; - String
;; - Symbol
```

Valid Racket Programs

- `(* (+ 1 2)`
  `(- 4 3))` ← Tree?
- `(* (+ 1 2)`
  `(- 4 3)`
  `(/ 10 5))`

Each tree "node" is a list, of … RacketProgs ??

But: how many values does each node have??

```
;; A RacketProg is a:
;; - Atom
;; - ProgTree
```

```
;; An Atom is a:
;; - Number
;; - String
;; - Symbol
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
```

Recursive Data Def!

Valid Racket Programs

Also, **Intertwined** Data Defs!

```
;; A RacketProg is a:
;; - Atom
;; - ProgTree
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
```

Intertwined Data

- A <u>set</u> of Data Definitions that reference each other
- <u>Templates</u> should be defined together …

```
;; A RacketProg is a:
;; - Atom
;; - ProgTree
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
```

Intertwined Data

- A set of Data Definitions that reference each other
- Templates should be defined together …
  - … and should reference each other's templates (when needed)

```
;; A RacketProg is one of:
;; - Atom
;; - ProgTree
(define (prog-fn p) ...)
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
(define (atom-fn a) ...)
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
(define (ptree-fn t) ...)
```

???

- Repo: **cs450f23/lecture16-inclass**
- File: **intertwined-template-**<*your last name*>**.rkt**

In-class Coding 11/1 #1: Intertwined Templates

- Templates should be defined together …
  - … and should reference each other's templates (when needed)

```
;; A RacketProg is one of:
;; - Atom
;; - ProgTree
(define (prog-fn p) ...)
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketPRog ProgTree)
(define (ptree-fn t) ...)
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
(define (atom-fn a) ...)
```

???

Intertwined Templates

```
;; A RacketProg is one of:
;; - Atom
;; - ProgTree
(define (prog-fn s)
  (cond
   [(list? s) ... (ptree-fn s) ...]
   [else      ... (atom-fn s) ...]))
```

Can swap cond ordering (to make distinguishing items easier)

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
(define (atom-fn a)
 (cond
   [(number? a) ... ]
   [(string? a) ... ]
   [else ... ]))
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
(define (ptree-fn t)
  (cond
   [(empty? t) ...]
   [else  ... (prog-fn (first t)) ... (ptree-fn (rest t)) ...]))
```

Intertwined data have intertwined templates!

“Racket Prog” = S-expression!

```
;; A Sexpr is one of:
;; - Atom
;; - ProgTree
```

```
(define (sexpr-fn s)
  (cond
   [(list? s) ... (ptree-fn s) ...]
   [else       ... (atom-fn s) ...]))
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons Sexpr ProgTree)
```

```
(define (ptree-fn t)
  (cond
   [(empty? t) ...]
   [else  ... (sexpr-fn (first t)) ... (ptree-fn (rest t)) ...]))
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
```

```
(define (atom-fn a)
 (cond
   [(number? a) ... ]
   [(string? a) ... ]
   [else ... ]))
```

- Repo: **cs450f23/lecture16-inclass**
- File: **count-symbol**-<*your last name*>.rkt

In-class Coding 11/1 #2: Counting Symbols

```
;; count : Symbol Sexpr -> Nat
;; Computes the number of times the given
;; symbol appears in the given s-expression
```

```
;; count-ptree : Symbol ProgTree -> Nat
;; ???
```

```
;; count-atom : Symbol Atom -> Nat
;; ???
```

No More Quizzes!